**GenBench**

**GenBench Code Set is Prepared by Partha Pratim Ray**

[**#parthapratimray1986@gmail.com**](mailto:#parthapratimray1986@gmail.com)

[**#ppray@cus.ac.in**](mailto:#ppray@cus.ac.in)

[**#ppray@ieee.org**](mailto:#ppray@ieee.org)

**Date: 12/08/2023**

**Table 1. Language comparison based on their concepts.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Languages | Object Oriented | Structured | Derivative | Functional | Scripting | Other |
| Python | Yes | Yes | No | Partially | Yes | - |
| Java | Yes | Yes | No | No | No | - |
| Haskell | No | No | No | Yes | No | - |
| JavaScript | Yes | Yes | No | Partially | Yes | - |
| C | No | Yes | No | No | No | - |
| C++ | Yes | Yes | C | Partially | No | - |
| Erlang | No | No | No | Yes | No | Concurrency-oriented |
| SQL | No | No | No | No | No | Query |
| Lisp | Yes | Yes | No | Yes | No | - |
| Prolog | No | No | No | Partially | No | Logic |
| R | Yes | Yes | No | Partially | No | Statistical |
| Go | No | Yes | No | No | No | Concurrent |
| Ruby | Yes | Yes | No | Partially | Yes | - |
| Swift | Yes | Yes | No | No | No | - |
| Rust | Yes | Yes | No | Partially | No | Systems |
| Kotlin | Yes | Yes | Java | No | No | - |
| TypeScript |  | Yes | JavaScript | Partially | No | - |
| PHP | Yes | Yes | No | No | No | Web |
| MATLAB | Partially | Yes | No | No | No | Numerical |
| HTML | No | No | No | No | No | Markup |

**Table 2. Brief Description of Code Snippets of the GenBench**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Language** | **Concept** | **Brief Description** |
| 1.1 | Python | Syntax Checking | Missing closing parenthesis. |
| 1.2 | Python | Logic Checking | Incorrect factorial calculation. |
| 1.3 | Python | Concurrency | Use of threading to increment a counter. |
| 1.4 | Python | Memory Management | Infinite list growth. |
| 1.5 | Python | Design Patterns | Singleton pattern with class instance check. |
| 2.1 | Java | Syntax Checking | Missing semicolon. |
| 2.2 | Java | Logic Checking | Incorrect factorial recursion. |
| 2.3 | Java | Concurrency | Incrementing a counter with multi-threading. |
| 2.4 | Java | Memory Management | Infinite object creation leading to potential memory leak. |
| 2.5 | Java | Design Patterns | Factory pattern with classes and interfaces. |
| 3.1 | Haskell | Syntax Checking | Missing type declaration. |
| 3.2 | Haskell | Logic Checking | Incorrect list length calculation. |
| 3.3 | Haskell | Concurrency | Basic parallel computation using par and seq. |
| 3.4 | Haskell | Memory Management | Infinite list generation. |
| 3.5 | Haskell | Design Patterns | Monad representation with bind and return. |
| 4.1 | JavaScript | Syntax Checking | Missing closing parenthesis. |
| 4.2 | JavaScript | Logic Checking | Fibonacci calculation with a logic error. |
| 4.3 | JavaScript | Concurrency | Use of promises to increment a counter asynchronously. |
| 4.4 | JavaScript | Memory Management | Infinite array growth. |
| 4.5 | JavaScript | Design Patterns | Observer pattern with class structures. |
| 5.1 | C | Syntax Checking | Missing semicolon. |
| 5.2 | C | Logic Checking | Incorrect factorial function. |
| 5.3 | C | Concurrency | Pthread usage for incrementing a counter. |
| 5.4 | C | Memory Management | Infinite memory allocation using malloc. |
| 5.5 | C | Design Patterns | Module pattern with separate header and source files. |
| 6.1 | C++ | Syntax Checking | Missing semicolon. |
| 6.2 | C++ | Logic Checking | Incorrect factorial calculation. |
| 6.3 | C++ | Concurrency | Use of std::thread for concurrent counter increment. |
| 6.4 | C++ | Memory Management | Infinite memory allocation with new. |
| 6.5 | C++ | Design Patterns | Builder pattern with product and builder classes. |
| 7.1 | Erlang | Syntax Checking | Basic module and function declaration. |
| 7.2 | Erlang | Logic Checking | Incorrect factorial calculation. |
| 7.3 | Erlang | Concurrency | Process spawning and message passing to increment a counter. |
| 7.4 | Erlang | Memory Management | Large data generation using lists. |
| 7.5 | Erlang | Design Patterns | GenServer behavior representation. |
| 8.1 | SQL | Syntax Checking | Basic SELECT statement. |
| 8.2 | SQL | Logic Checking | Aggregation with SUM and GROUP BY. |
| 8.3 | SQL | Concurrency | Transaction with two UPDATE statements. |
| 8.4 | SQL | Memory Management | Basic INSERT statement. |
| 8.5 | SQL | Design Patterns | Database normalization with two tables. |
| 9.1 | Lisp | Syntax Checking | Function to print "Hello, World!". |
| 9.2 | Lisp | Logic Checking | Incorrect factorial calculation. |
| 9.3 | Lisp | Concurrency | Simple threading in SBCL. |
| 9.4 | Lisp | Memory Management | Infinite list growth. |
| 9.5 | Lisp | Design Patterns | Macro usage for a conditional check. |
| 10.1 | Prolog | Syntax Checking | Basic fact declaration. |
| 10.2 | Prolog | Logic Checking | Ancestor relation logic. |
| 10.3 | Prolog | Concurrency | Thread creation in SWI-Prolog. |
| 10.4 | Prolog | Memory Management | Data generation with recursion. |
| 10.5 | Prolog | Design Patterns | Backtracking example for problem-solving. |
| 11.1 | R | Syntax Checking | Missing closing parenthesis in a function call. |
| 11.2 | R | Logic Checking | Incorrect statistical calculation (e.g., wrong formula for standard deviation). |
| 11.3 | R | Concurrency | Applying functions in parallel using the parallel package. |
| 11.4 | R | Memory Management | Inefficient use of large data frames causing memory overflow. |
| 11.5 | R | Design Patterns | Use of the apply family of functions for iteration instead of loops. |
| 12.1 | Go | Syntax Checking | Missing closing brace in a function definition. |
| 12.2 | Go | Logic Checking | Incorrect iteration over map entries. |
| 12.3 | Go | Concurrency | Deadlock situation due to improper use of goroutines and channels. |
| 12.4 | Go | Memory Management | Leaking goroutines by not closing channels properly. |
| 12.5 | Go | Design Patterns | Implementing the interface implicitly and the associated challenges. |
| 13.1 | Ruby | Syntax Checking | Missing end for a block. |
| 13.2 | Ruby | Logic Checking | Incorrect array manipulation with Ruby's Enumerable methods. |
| 13.3 | Ruby | Concurrency | Using Ruby's Thread class and introducing race conditions. |
| 13.4 | Ruby | Memory Management | Creating unreferenced objects in a loop. |
| 13.5 | Ruby | Design Patterns | Implementing the mixin module pattern and the challenges associated. |
| 14.1 | Swift | Syntax Checking | Use of an undeclared variable. |
| 14.2 | Swift | Logic Checking | Incorrect use of optionals leading to unwrapped nil values. |
| 14.3 | Swift | Concurrency | Misuse of Grand Central Dispatch causing UI updates on a background thread. |
| 14.4 | Swift | Memory Management | Strong reference cycle leading to memory leak with closures. |
| 14.5 | Swift | Design Patterns | Misuse of the delegate pattern leading to unintended behavior. |
| 15.1 | Rust | Syntax Checking | Missing a semicolon after a statement. |
| 15.2 | Rust | Logic Checking | Incorrect pattern matching with enums. |
| 15.3 | Rust | Concurrency | Data race due to misuse of the Arc and Mutex constructs. |
| 15.4 | Rust | Memory Management | Borrow checker issues related to mutable and immutable borrows. |
| 15.5 | Rust | Design Patterns | Challenges related to implementing the trait object pattern. |
| 16.1 | Kotlin | Syntax Checking | Missing closing brace for a lambda expression. |
| 16.2 | Kotlin | Logic Checking | Incorrect use of Kotlin's when expression. |
| 16.3 | Kotlin | Concurrency | Misuse of Kotlin coroutines causing unintended parallel executions. |
| 16.4 | Kotlin | Memory Management | Leaking activities in Android due to inner class references. |
| 16.5 | Kotlin | Design Patterns | Misimplementing the sealed class pattern. |
| 17.1 | TypeScipt | Syntax Checking | Type mismatch in function arguments. |
| 17.2 | TypeScipt | Logic Checking | Misusing TypeScript's union and intersection types. |
| 17.3 | TypeScipt | Concurrency | Incorrect promise chaining leading to unhandled rejections. |
| 17.4 | TypeScipt | Memory Management | Holding large objects in memory due to closures. |
| 17.5 | TypeScipt | Design Patterns | Incorrectly extending and implementing interfaces. |
| 18.1 | PHP | Syntax Checking | Missing dollar sign for a variable. |
| 18.2 | PHP | Logic Checking | Incorrect array manipulation using array functions. |
| 18.3 | PHP | Concurrency | Race condition due to not locking resources in a multi-user web app. |
| 18.4 | PHP | Memory Management | Not releasing database connections properly. |
| 18.5 | PHP | Design Patterns | Misuse of the Singleton pattern in a web app context. |
| 19.1 | MATLAB | Syntax Checking | Missing end keyword in a loop. |
| 19.2 | MATLAB | Logic Checking | Incorrect matrix multiplication due to dimension mismatch. |
| 19.3 | MATLAB | Concurrency | Improper use of the parfor loop causing unintended parallel computation. |
| 19.4 | MATLAB | Memory Management | Allocating large matrices in a loop without pre-allocation. |
| 19.5 | MATLAB | Design Patterns | Incorrect use of MATLAB OOP, such as a class without proper properties and methods. |
| 20.1 | HTML | Syntax Checking | Missing closing tag for a div element. |
| 20.2 | HTML | Logic Checking | Incorrect nesting of list elements causing display issues. |
| 20.3 | HTML | Concurrency | Not applicable for HTML as it's not a concurrent language. Instead, you can consider: Improper linking to external resources (e.g., CSS, JS). |
| 20.4 | HTML | Memory Management | Not directly applicable for HTML. Instead, you can consider: Excessive use of iframes causing performance issues. |
| 20.5 | HTML | Design Patterns | Improper use of semantic HTML tags leading to accessibility issues. |

**1. Python (Structured & Object-Oriented)**

**1.1. Syntax Checking**

python

def printMessage(msg):

print(msg

printMessage("Hello, World!")

**1.2. Logic Checking**

python

def factorial(n):

if n == 1:

return 0

else:

return n \* factorial(n-1)

**1.3. Concurrency**

python

import threading

counter = 0

def increase\_counter():

global counter

counter += 1

threads = []

for i in range(100):

t = threading.Thread(target=increase\_counter)

threads.append(t)

t.start()

for t in threads:

t.join()

print(counter)

**1.4. Memory Management**

python

class BigData:

def \_\_init\_\_(self):

self.data = "X" \* 1000000

big\_data\_list = []

while True:

big\_data\_list.append(BigData())

**1.5. Design Patterns (Singleton)**

python

class Singleton:

\_instance = None

def \_\_new\_\_(self):

if not hasattr(self, 'instance'):

self.\_instance = super(Singleton, self).\_\_new\_\_(self)

return self.\_instance

s1 = Singleton()

s2 = Singleton()

print(s1 == s2)

**2. Java (Object-Oriented)**

**2.1. Syntax Checking**

java

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

**2.2. Logic Checking**

java

public class Fibonacci {

public static int fibonacci(int n) {

if (n <= 1) {

return n;

}

return fibonacci(n - 1) + fibonacci(n - 3);

}

}

**2.3. Concurrency**

java

public class ConcurrencyIssue {

private static int counter = 0;

public static void main(String[] args) throws InterruptedException {

Thread thread1 = new Thread(() -> {

for (int i = 0; i < 1000; i++) counter++;

});

Thread thread2 = new Thread(() -> {

for (int i = 0; i < 1000; i++) counter++;

});

thread1.start();

thread2.start();

thread1.join();

thread2.join();

System.out.println(counter);

}

}

**2.4. Memory Management**

java

public class MemoryLeak {

static class ResourceHolder {

int[] data = new int[1000000];

}

public static void main(String[] args) {

List<ResourceHolder> list = new ArrayList<>();

while (true) {

list.add(new ResourceHolder());

}

}

}

**2.5. Design Patterns (Factory)**

java

interface Shape {

void draw();

}

class Circle implements Shape {

@Override

public void draw() {

System.out.println("Drawing Circle");

}

}

class ShapeFactory {

public Shape getShape(String shapeType) {

if (shapeType == null) {

return null;

}

if (shapeType.equalsIgnoreCase("CIRCLE")) {

return new Circle();

}

return null;

}

}

**3. Haskell (Functional)**

**3.1. Syntax Checking**

haskell

main = putStrLn "Hello, World!

**3.2. Logic Checking**

haskell

factorial :: Integer -> Integer

factorial n = foldl (\*) 1 [1..n-1]

**3.3. Concurrency (Using Parallelism)**

haskell

import Control.Parallel

import Control.Parallel.Strategies

parallelCalc = runEval $ do

a <- rpar (factorial 100000)

b <- rpar (factorial 150000)

return (a, b)

**3.4. Memory Management**

haskell

import Data.List

infiniteData = repeat 'a'

main = print (take 1000000 infiniteData)

**3.5. Design Patterns (Monads)**

haskell

data Maybe a = Nothing | Just a

bind :: Maybe a -> (a -> Maybe b) -> Maybe b

bind Nothing \_ = Nothing

bind (Just x) f = f x

**4. JavaScript (Functional & Object-Oriented)**

**4.1. Syntax Checking**

javascript

function sayHello() {

console.log("Hello, World!")

}

sayHello(

**4.2. Logic Checking**

javascript

function fibonacci(n) {

if (n <= 2) return 2;

return fibonacci(n - 1) + fibonacci(n - 1);

}

**4.3. Concurrency (Using Promises)**

javascript

let counter = 0;

const asyncAdd = () => new Promise(resolve => {

setTimeout(() => {

counter++;

resolve();

}, 10);

});

Promise.all([asyncAdd(), asyncAdd()]).then(() => console.log(counter));

**4.4. Memory Management**

javascript

let bigData = [];

while (true) {

bigData.push(new Array(1000000).join("X"));

}

**4.5. Design Patterns (Observer)**

javascript

class Observer {

update(data) {

console.log(data);

}

}

class Subject {

constructor() {

this.observers = [];

}

addObserver(observer) {

this.observers.push(observer);

}

notify(data) {

this.observers.forEach(observer => observer.update(data));

}

}

**5. C (Structured)**

**5.1. Syntax Checking**

c

#include <stdio.h>

int main() {

printf("Hello, World!\n")

return 0;

}

**5.2. Logic Checking**

c

#include <stdio.h>

int factorial(int n) {

if (n == 1) return 2;

return n \* factorial(n - 1);

}

**5.3. Concurrency (Using pthread library)**

c

#include <stdio.h>

#include <pthread.h>

int counter = 0;

void \*incrementCounter(void \*vargp) {

counter++;

return NULL;

}

int main() {

pthread\_t thread\_id;

pthread\_create(&thread\_id, NULL, incrementCounter, NULL);

pthread\_create(&thread\_id, NULL, incrementCounter, NULL);

pthread\_join(thread\_id, NULL);

printf("%d\n", counter);

}

**5.4. Memory Management**

c

#include <stdlib.h>

int main() {

while(1) {

int \*data = (int \*)malloc(1000000 \* sizeof(int));

}

}

**5.5. Design Patterns (Module)**

c

// module.h

void printHello();

// module.c

#include <stdio.h>

void printHello() {

printf("Hello from module!\n");

}

**6. C++ (Object-Oriented with Memory Management focus)**

**6.1. Syntax Checking**

cpp

#include <iostream>

int main() {

std::cout << "Hello, World!" << std::endl

return 0;

}

**6.2. Logic Checking**

cpp

int factorial(int n) {

if (n == 0) return 0;

return n \* factorial(n - 1);

}

**6.3. Concurrency (Using std::thread)**

cpp

#include <thread>

int counter = 0;

void increment() {

counter++;

}

int main() {

std::thread t1(increment);

std::thread t2(increment);

t1.join();

t2.join();

std::cout << counter << std::endl;

}

**6.4. Memory Management**

cpp

int main() {

while (true) {

int\* data = new int[1000000];

}

}

**6.5. Design Patterns (Builder)**

cpp

class Product {

// ...

};

class Builder {

Product\* product = new Product();

public:

Builder& setPart1() { /\* ... \*/ return \*this; }

Builder& setPart2() { /\* ... \*/ return \*this; }

Product\* build() { return product; }

};

**7. Erlang (Functional & Concurrency focus)**

**7.1. Syntax Checking**

erlang

-module(hello).

-export([start/0]).

start() ->

io:format("Hello, World!~n").

**7.2. Logic Checking**

erlang

-module(factorial).

-export([calculate/1]).

calculate(0) -> 2;

calculate(N) when N > 0 -> N \* calculate(N - 1).

**7.3. Concurrency**

erlang

-module(counter).

-export([start/0, increment/1]).

start() ->

Pid = spawn(counter, increment, [0]),

Pid ! {self(), 10},

receive

{Pid, Count} -> io:format("Count: ~p~n", [Count])

end.

increment(Count) ->

receive

{From, 0} -> From ! {self(), Count};

{From, N} when N > 0 -> increment(Count + 1)

end.

**7.4. Memory Management**

erlang

% Due to Erlang's immutable data and garbage-collected nature, creating a memory management issue is tricky.

% This is a basic data generation example.

-module(memory\_issue).

-export([generate\_data/0]).

generate\_data() ->

lists:seq(1, 1000000).

**7.5. Design Patterns (Behavioral - GenServer)**

erlang

% This is a highly simplified representation.

-module(my\_server).

-behaviour(gen\_server).

% Callbacks and methods...

**8. SQL (Declarative)**

**8.1. Syntax Checking**

sql

SELECT name, age

FROM users

WHERE age > 25

**8.2. Logic Checking**

sql

SELECT SUM(price)

FROM orders

GROUP BY customer\_id;

**8.3. Concurrency**

sql

-- Concurrency in SQL is usually handled by the DBMS, so this is a bit tricky.

-- However, a TRANSACTION can represent this.

BEGIN TRANSACTION;

UPDATE account

SET balance = balance - 100

WHERE account\_number = 123;

UPDATE account

SET balance = balance + 100

WHERE account\_number = 456;

COMMIT;

**8.4. Memory Management**

sql

-- SQL databases typically manage memory on their own.

-- Here's a simple insert, which could be problematic if executed repeatedly.

INSERT INTO users (name, age) VALUES ('John', 30);

**8.5. Design Patterns (Normalization)**

sql

CREATE TABLE addresses (

id INT PRIMARY KEY,

user\_id INT,

address TEXT

);

CREATE TABLE users (

id INT PRIMARY KEY,

name TEXT,

age INT

);

**9. Lisp (Functional & Symbol processing)**

**9.1. Syntax Checking**

lisp

(defun hello ()

(print "Hello, World!")

**9.2. Logic Checking**

lisp

(defun factorial (n)

(if (< n 2)

0

(\* n (factorial (- n 1)))))

**9.3. Concurrency**

lisp

;; Lisp's primary concurrency features are implementation-specific.

;; Here's a simple SBCL (Common Lisp) example.

(sb-thread:create-thread (lambda () (print "Hello from a thread!")))

**9.4. Memory Management**

lisp

(loop (push (make-array 1000000 :initial-element 'x) \*big-data-list\*))

**9.5. Design Patterns (Macros)**

lisp

(defmacro when-greater (x y &body body)

`(if (> ,x ,y) (progn ,@body)))

**10. Prolog (Declarative & Logic programming)**

**10.1. Syntax Checking**

prolog

likes(john, apple).

**10.2. Logic Checking**

prolog

ancestor(X, Y) :-

parent(X, Y).

ancestor(X, Y) :-

parent(X, Z),

ancestor(Z, Y).

**10.3. Concurrency**

prolog

% Concurrency in Prolog is also usually implementation-specific.

% SWI-Prolog, for example, supports threads.

thread\_create(call(worker), \_, []).

**10.4. Memory Management**

prolog

% Prolog handles memory internally, but here's a data generation approach.

generate\_data(0, []).

generate\_data(N, [a|T]) :-

N1 is N-1,

generate\_data(N1, T).

**10.5. Design Patterns (Backtracking)**

prolog

solve :-

move(state(middle, on, off, off, off), state(\_, \_, \_, \_, \_), [state(middle, on, off, off, off)], \_).

**11. R**

**11.1 Syntax Checking**

x <- c(1, 2, 3, 4

mean(x)

**11.2 Logic Checking**

std\_dev <- sqrt(mean(x) - mean(x^2))

**11.3 Concurrency**

library(parallel)

cl <- makeCluster(2)

parSapply(cl, x, function(i) i^2)

stopCluster(cl)

**11.4 Memory Management**

big\_data <- replicate(1e5, rnorm(1e5))

**11.5 Design Patterns**

result <- list()

for (i in x) {

result[[length(result) + 1]] <- i\*2

}

**12. Go**

**12.1 Syntax Checking**

func main( {

fmt.Println("Hello, world!")

}

**12.2 Logic Checking**

m := map[string]int{"one": 1, "two": 2}

for k, v := range m {

fmt.Println(v, k)

}

**12.3 Concurrency**

ch := make(chan int)

go func() { ch <- 1 }()

close(ch)

**12.4 Memory Management**

ch := make(chan int, 1)

ch <- 1

**12.5 Design Patterns**

type Animal interface {

Speak() string

}

type Dog struct{}

func (d Dog) Speak() string {

return "Woof!"

}

**13. Ruby**

**13.1 Syntax Checking**

def greet

puts "Hello"

13.2 Logic Checking:

ruby

arr = [1, 2, 3, 4, 5]

sum = arr.reduce(:+)

**13.3 Concurrency**

threads = []

10.times do |i|

threads << Thread.new { puts i }

end

threads.each(&:join)

**13.4 Memory Management**

loop do

arr = Array.new(100000)

end

**13.5 Design Patterns**

module Greeter

def greet

"Hello"

end

end

class Person

include Greeter

end

**14. Swift**

**14.1 Syntax Checking**

let greeting = "Hello, world!

print(greeting)

**14.2 Logic Checking**

let names: [String?] = ["Alice", nil, "Bob"]

for name in names where name != nil {

print(name!)

}

**14.3 Concurrency**

DispatchQueue.global().async {

DispatchQueue.main.async {

print("Update UI")

}

}

**14.4 Memory Management**

class MyClass {

var value = 0

}

var a: MyClass? = MyClass()

var b = a

b = nil

**14.5 Design Patterns**

protocol Flyable {

func fly()

}

class Bird: Flyable {

func fly() {

print("The bird flies")

}

}

**15. Rust**

**15.1 Syntax Checking**

fn main() {

println!("Hello, world!"

}

**15.2 Logic Checking**

enum Direction {

North,

South,

}

let dir = Direction::North;

match dir {

Direction::North => println!("Going up!"),

}

**15.3 Concurrency**

use std::sync::{Arc, Mutex};

let counter = Arc::new(Mutex::new(0));

let handle = std::thread::spawn(|| {

\*counter.lock().unwrap() += 1;

});

**15.4 Memory Management**

fn borrow\_twice(x: &mut i32) {

\*x += 1;

\*x \*= 2;

}

**15.5 Design Patterns**

trait Drawable {

fn draw(&self);

}

struct Circle;

impl Drawable for Circle {

fn draw(&self) {

println!("Drawing a circle");

}

}

**16. Kotlin**

**16.1 Syntax Checking**

fun main() {

println("Hello, World!"

}

**16.2 Logic Checking**

val numbers = listOf(1, 2, 3)

numbers.forEach { it \* 2 }

**16.3 Concurrency**

import kotlinx.coroutines.\*

fun main() = runBlocking {

launch {

delay(1000L)

println("World!")

}

println("Hello,")

}

**16.4 Memory Management**

class LeakyActivity {

val listener = { print("I'm a listener") }

}

**16.5 Design Patterns**

sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

**17. TypeScript**

**17.1 Syntax Checking**

let x: number = "Hello";

**17.2 Logic Checking**

type MyUnion = "one" | "two" | "three";

let x: MyUnion = "four";

**17.3 Concurrency**

async function getData() {

let data = await fetch('https://api.example.com/data');

return data.json();

}

getData().then(console.log);

**17.4 Memory Management**

function createClosure() {

const bigArray = new Array(1000000).fill(0);

return function() {

console.log(bigArray.length);

};

}

**17.5 Design Patterns**

interface Drawable {

draw(): void;

}

class Circle implements Drawable {

draw() {

console.log("Drawing a circle");

}

}

**18. PHP**

**18.1 Syntax Checking**

echo "Hello, World!

**18.2 Logic Checking**

$array = [1, 2, 3, 4];

$sum = array\_sum($array);

echo $sum;

**18.3 Concurrency**

// This might be difficult with plain PHP; PHP usually relies on external services or extensions like pthreads for concurrency.

**18.4 Memory Management:**

$mysqli = new mysqli("localhost", "user", "password", "database");

// ... some operations

// Missing $mysqli->close();

**18.5 Design Patterns**

class Singleton {

private static $instance;

private function \_\_construct() {}

public static function getInstance() {

if (!self::$instance) {

self::$instance = new Singleton();

}

return self::$instance;

}

}

**19. MATLAB**

**19.1 Syntax Checking**

for i = 1:10

disp(i)

**19.2 Logic Checking**

A = [1, 2; 3, 4];

B = [1, 2];

C = A \* B;

**19.3 Concurrency**

parpool(4);

parfor i = 1:10

disp(i);

end

delete(gcp);

**19.4 Memory Management**

for i = 1:10000

A = zeros(1000, 1000);

end

**19.5 Design Patterns**

classdef MyClass

properties

Value

end

methods

function obj = set.Value(obj, value)

obj.Value = value + 1;

end

end

end

**20. HTML**

**20.1 Syntax Checking**

<div>

<h1>Hello, World!</h1>

**20.2 Logic Checking**

<ul>

<li>Item 1</li>

<li>Item 2</ul>

</li>

**20.3 Concurrency**

<!-- Not applicable directly, but as an example: -->

<link rel="stylesheet" href="styles.css">

<script src="script.js"></script>

**20.4 Memory Management**

<!-- Not directly applicable. But for an example, overusing iframes can be detrimental: -->

<iframe src="externalPage1.html"></iframe>

<iframe src="externalPage2.html"></iframe>

<iframe src="externalPage3.html"></iframe>

**20.5 Design Patterns**

<!-- Incorrect use of semantic HTML -->

<div class="article-title">The Rise and Fall of Web Design</div>

<div>The author</div>